ENAE788M Project 3a
Team Bouncing Rainbow Zebras

Erik Holum
Graduate Student
University of Maryland
Email: eholum@gmail.com

Edward Carney
Graduate Student
University of Maryland
Email: carneyedwardj@gmail.com

Derek Thompson
Graduate Student
University of Maryland
Email: derekbt@yahoo.com

***Abstract*—We examine the problem of flying a Bebop Quadrotor through a gate at unknown position relative the takeoff location of the drone. We first test using several methods of color segmentation for gate recognition, including thresholding, single Gaussians, and Gaussian Mixture models; then discuss the process of identifying the position and orientation of the gate using camera feedback and the OpenCV Library. Given an approximate position, we present a method of producing a filtered reading, then results of testing the control algorithm.**

I. Introduction

In this project, our aim was to develop a computer vision feedback system capable of autonomously flying our Bebop drone through a yellow gate at unknown starting position and orientation. For the purposes of this goal, we assume that the gate is at least in partial view of the front facing camera at takeoff, so we will do not require any kind of search algorithm to locate the gate at start.

The first task was to evaluate several different methods for doing color segmentation on image data, including thresholding, single Gaussians, and Gaussian Mixture models (GMMs). We provide an overview of our process and results, and discuss the many pitfalls we discovered in collecting image data and training our classifiers.

Given the output of the color segmentation algorithm, we use the tools supplied by the OpenCV Library [1] to find the corners of the gate. Once the corners have been identified, we leverage $solvePnP$ and precomupted camera calibration information to deduce the location of the gate relative the quadrotor's body fixed frame.

Unless otherwise specified, all software was implemented using ROS [2] and Python.

II. Color Segmentation

In this section we discuss our approach for implementing, training, and testing various methods of color segmentation. The first step in the process was to gather a significant amount of test data for training and testing our implementations. We used rosbag to record images of both yellow and purple gate for a variety of lighting conditions (day vs. night, different levels of illuminations settings for overhead lighting). We then used Matlab's $roipoly$ function to create mask files to remove any non-interesting pixels from images selected from our bag files, as in Figure 1.

Once we had test images and masks, we extracted all relevant pixels into a single numpy array and were able to save the data for training in a convienient '.npy' file. We then examined different color encodings offered through OpenCSV to see which schemes gave us the most 'suitable' shapes for fitting Gaussians or thresholding. In general, we found BGR, HSV, and LAB schemes to be particularly useful. A 3d plot of all training data is presented in Figure 2.

A. Thresholding

The first, and simplest method of color segmentation is Thresholding. For each pixel $[C_1, C_2, C_3]$, where C_i is a measurement specific to a specific color scheme, we simply set limits, ϵ_i, to determine if a pixels is 'yellow'. i.e., $C_i > \epsilon_i$ for $i = 1, 2, 3$. While the simple lower bound produced decent results, we found that bounding each axis both above and below proved slightly easier. In particular, for let μ_i and σ_i be the mean value and standard deviation of a training set along the i^{th} axis, then our test becomes:

$$\sigma_i \geq |\mu_i - C_i|.$$

In other words, if each axis is within one standard deviation of the mean. To test the accuracy of our thresholder classifiers, we applied the test to all of the images in the training set, then computed two values:

1) The total percentage of correct pixels over the entire image, where correctness is defined by our masks.
2) The total percentage of 'false negatives' in the masked region. In other words, what percentage of the masked values did the classifier correctly identify as yellow?

The second test was particularly informative, since false negatives are in some way more detrimental towards gate identifications (at least in our experience). The plots for different color scheme's performances are presented in Figure 4. As evidenced by the plots, HSV, YUV, and LAB provided the best overall 'performance'. However, regardless of the color schemes, the images ended up being very noisy. As demonstrated in Figure 3.

Overall, the simple thresholder performed adequately on medium lighting, but poorly at the extremes. Which given the bounds of σ_i for the entire training set, makes perfect sense.

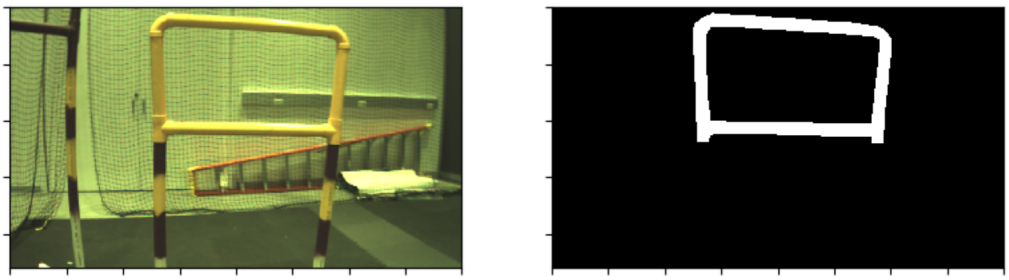

Fig. 1. Left: Raw image of yellow at 800x460 resolution. Right: Mask generated in Matlab using $roipoly$.

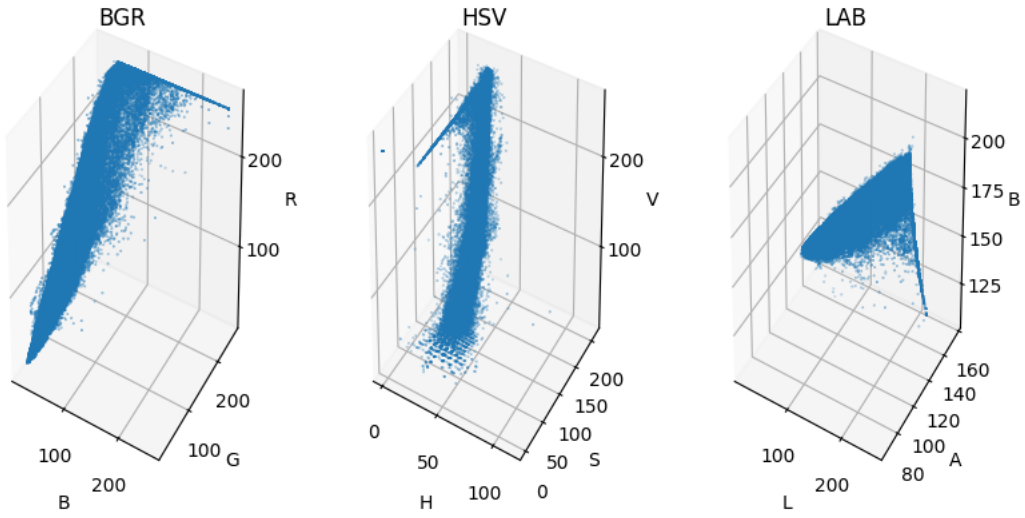

Fig. 2. Training data for yellow pixels visualized under three different color schemes. Left to right: BGR, HSV, and LAB.

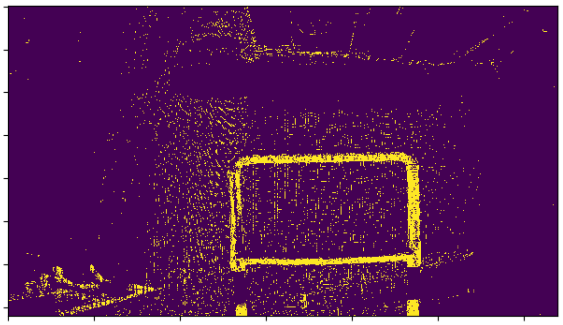

Fig. 3. Result of testing a threshold HSV classifier on a yellow gate.

B. Single Gaussian

We next implemented and testing using a single Gaussian based on the means, σ_i, and the covariance, A of the training data. We compute the likelihood for each pixel, x, without normalizing,

$$P(x|c_i) = \exp\left[-\frac{1}{2}(x-\mu)^T A^{-1}(x-\mu)\right].$$

We set a standard limit of 70% likelihood. A sample image classification is depicted in Figure 5. Note the drastically reduced noise.

Repeating the testing process from the threshold classifiers - the results are presented in Figure 6. Note that YCrCb provided both the highest noise, but also the most 'solid' false negative performance on the actual gate.

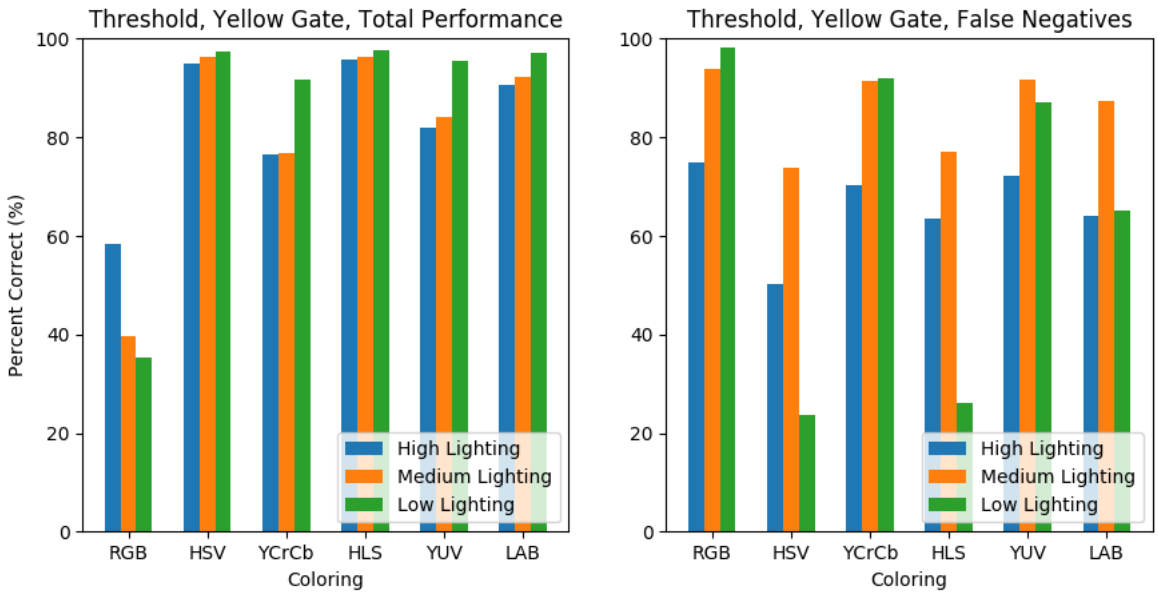

Fig. 4. Thresholding classifier performances for different color schemes. Total percentage correct on the left, false negative correctness on the right.

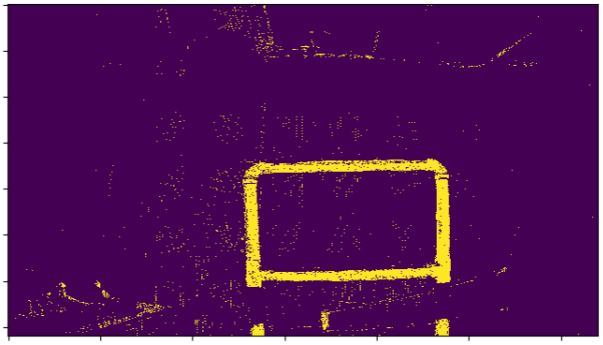

Fig. 5. Result of testing a single Gaussian YCrCb classifier on a yellow gate.

C. GMM

Finally, we implement an color segmentation tool that using a trained Gaussian Mixture Model (GMM). This included implementation of an EM algorithm to converge to optimal weights π_i, means σ_i, and covariances A_i to fit our data. Likelihood is computed with,

$$P(x|c_i) = \sum_{i=1}^{k} \pi_k \sqrt{\frac{\det A}{(2\pi)^3}} \exp\left[-\frac{1}{2}(x-\mu)^T A^{-1}(x-\mu)\right].$$

The results of training our GMM using the BGR, HSV, and LAB color schemes for k=2, k=3, and k=5, respectively, are demonstrated in Figure 7. Note, we only plot a small subset (5%) of the training data so that the Gaussians are more clearly visible, but the fit to the data is also apparent.

Overall, the GMMs significantly outperformed both other means of segmentation across all levels of lighting. We even added an additional level, morning daylight with maximum illumination, and found we were still able to get accurate results. A sample run of the classifier is provided in Figure 8. Note the significantly reduced noise over the previous classifiers.

The GMM results for all color schemes at multiple lighting levels are given in Figure **??**. The LAB and HSV color schemes performed particularly well. The values for k, σ_i, and $_i$ were all saved to files for quick and easy testing with the next steps of our project.

III. Gate Position and Orientation

This section reviews calibration of the Bebop's forward-facing camera, the methods used to process the masked images produced from the color segmentation, and the process to determine the estimated gate position and orientation from this data.

A. Camera Calibration

Camera calibration was performed manually with the assistance of the open-source Kalibr toolbox. Manual calibration included configuring RGB value offsets in the camera driver via color wheel testing and increasing/decreasing the camera exposure level. Although significant effort was expended adjusting the RGB offsets, for our test environment, lighting conditions, and camera, we found that the zeroing these offsets out and using the raw images for training and processing yielded the best results. On the other hand, the manipulation of the camera exposure levels helped to significantly improve image masking in various lighting conditions and allowed us

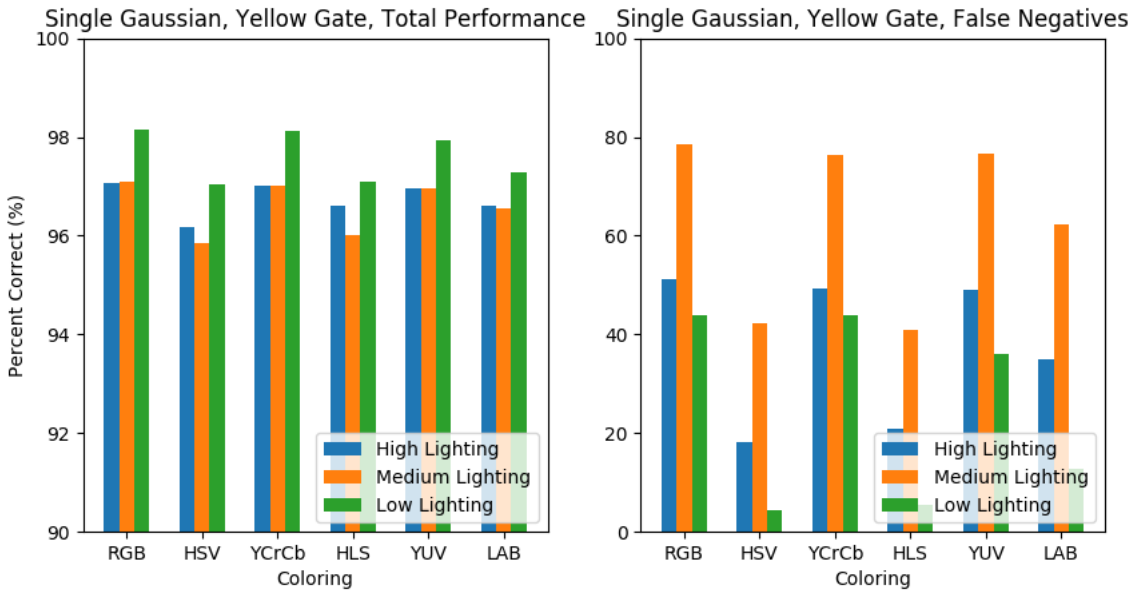

Fig. 6. Single Gaussian classifier performances for different color schemes. Total percentage correct on the left, false negative correctness on the right. Note the different limits on the y-axes.

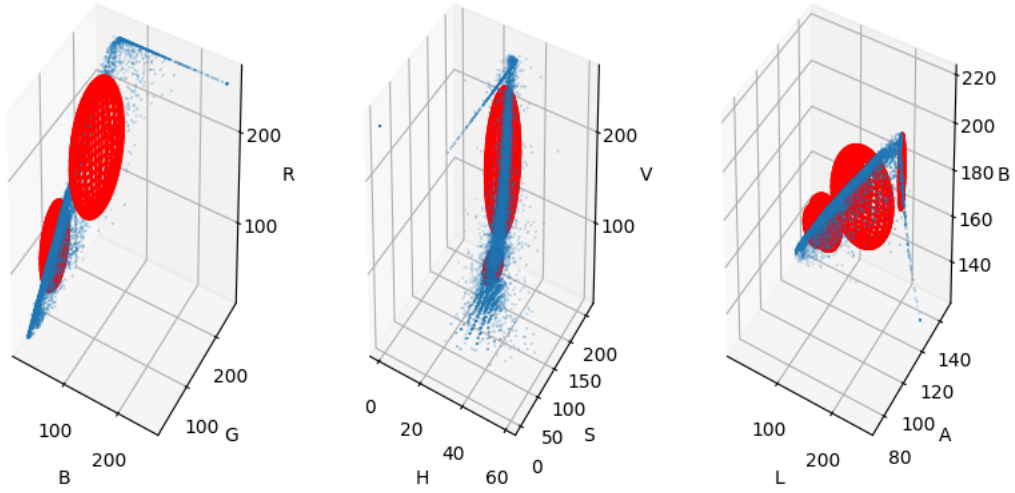

Fig. 7. GMM fits. Note only 5% of training data is shown for clearer Gaussians. Gaussians radii at 1-σ. Left to right: (BGR, k=2), (HSV, k=3), and (LAB, k=5).

Fig. 8. Result of testing a GMM, HSV with K=3 classifier on a yellow gate.

(in some cases) to improve the quality of the mask on-the-fly with some basic image testing.

The open-source Kalibr toolbox allowed us to easily estimate our camera's principle point and calibration matrix K. We collected videos at each of our camera's resolutions and processed each video using the toolbox. The resultant matrices and principle points were used in the estimation of gate orientation and position.

B. Post-Masking Processing

The masked images produced from the color thresholding were post-processed to extract the information necessary to determine the position and orientation of the gate. This process included extracting edges via Canny edge determination, finding line segments from the given edges via a probabilistic Hough transformation, calculating the intersection points of the Hough lines, and determining the four primary intersection point clusters (i.e. the gate edges) via K-Means clustering.

Prior to any other operations, the masked imaged was processed by removing occupied regions of the image that did not meet a minimum specified size. We experimented with some basic dilation and erosion operations, both before, after, and in-between small area removal, and this proved effective at further refining the image when run on full resolution masked images. However, erosion and dilation appeared less effective with the scaled-down images and ended up removing too much of the gate region, so only small area removal was used to process images prior to the Canny edge detection.

OpenCV's Canny edge detector was used to extract edges from the processed masked images. The minimum and maximum threshold values for this operation were set to be lenient, as the small area removal prior to edge detection removed the majority of image noise and greater effort was focused on the determination of relevant line segments via a Hough transform.

OpenCV's Probabilistic Hough transformation was used to extract line segments from the Canny edge images. Significant tuning was done with the minimum required line length and number of intersections and the maximum line gap parameters to eliminate as many errant line segments as possible. This reduced the number of segments needed to process for intersection determination (an $O(n^2)$ operation). Additionally, the line segments were post-processed via an extension algorithm to increase the line length and ensure that there were sufficient intersections to determine gate corner position.

Every line segments was then processed to extract all unique intersection points and NumPy's K Means clustering function was used to determine the four main point clusters within the image. All lines were checked for intersections, although included in each check was a step to compare the orientation of the lines relative to each other. This was done by comparing the line normal vectors, if the difference between the line normals was not significant (implying that the two lines were not close to perpendicular) any intersections between the two lines were ignored. This allowed us to throw out intersection points that may erroneously offset the K Means clustering results. As it was easily possible to obtain images where the intersection points only clustered in two or three areas (e.g. only a portion of the gate is within view), a post-processing step was run on the cluster regions to determine if any of the points were within a set radius of another cluster point; in these cases it was likely the full gate was not within view and the resultant gate position and orientation estimation would yield erroneous results, therefore these data were ignored.

Figures 1 and 12 show the above process visually on a sample image from testing.

C. Getting Position and Orientation

With the gate position relative to the vehicle's body frame, the gate frame in the world frame could be computed. The relative position from the body frame was projected from the vehicle's current odometry to get a position in the world frame. The projection only used the yaw from the vehicle's current attitude as the vehicle does not make aggressive enough maneuvers for the pitch and roll to significantly contribute to the projected gate position.

Once the was computed the gate positions were filtered to get a final estimated gate position. A log of the last n gate position and heading readings is maintained during the gate navigation. For our trails we used a log length, n, of 20 readings. While a gate position or heading is not confirmed, the filtered calculates the standard deviation for the position or heading reaches a low enough threshold the gate location or heading is confirmed. To filter out bad measurements another threshold is used to discard bad data. If the new gate reading is further from the current estimated gate position or heading then said threshold, the reading is discarded.

IV. State Logic

In order to control what the drone was doing at various steps in going through a gate, logic to control the process was created. The states follow a linear progression from the initial state to a defined final state. Each state contains information on what the drone needs to be doing at during this state and what conditions need to be met in order to move to the next state. The states are initialized as follows.

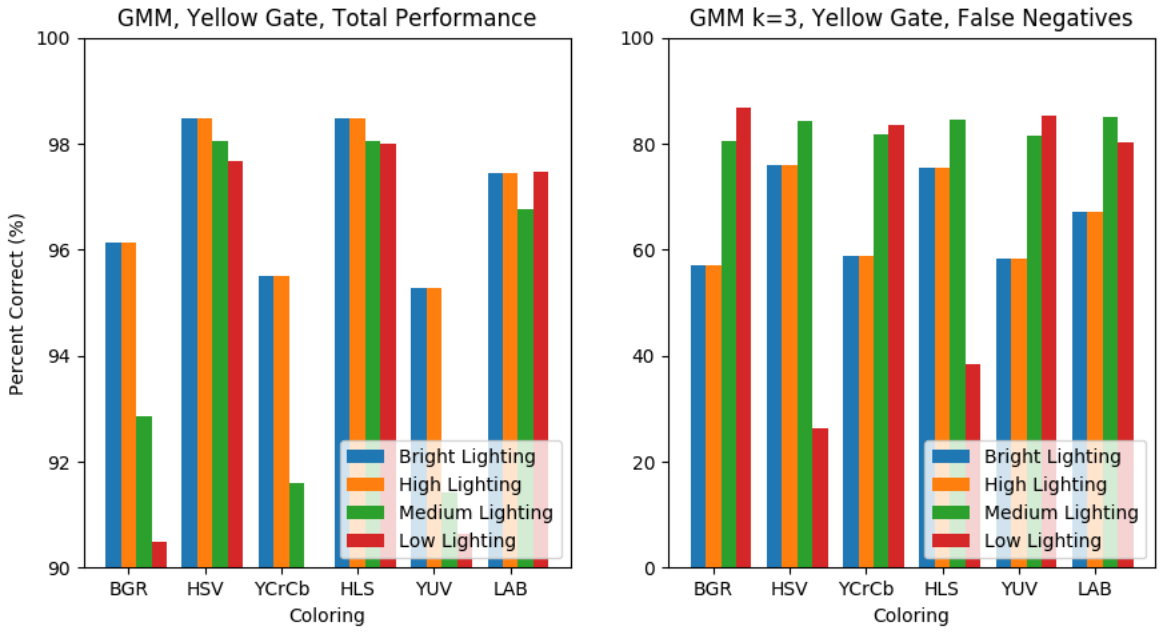

Fig. 9. GMM classifier performance on training data for k=3. The GMM method of classification was far more accurate that either other tested.

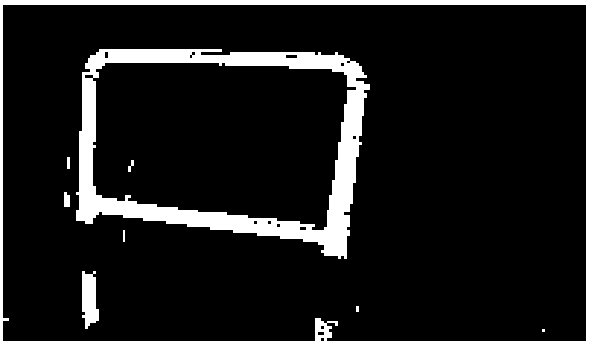

Fig. 10. Masked Gate Image

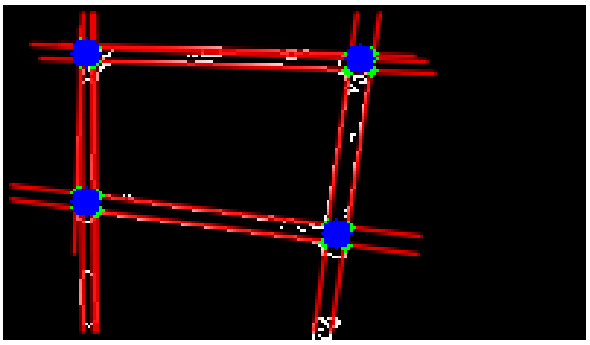

Fig. 11. Gate Image with Post-Processing and Corner Detection

$$
\begin{aligned}
states[n] = States(&own\_State\_Number,\\
&next\_States\_Number,\\
&exit\_Condition\_Type,\\
&exit\_Condition\_Threshold,\\
&gate\_Detection\_Active,\\
&controller\_Type,\\
&fly\_WP\_Location,\\
&look\_WP\_Location)
\end{aligned}
\tag{1}
$$

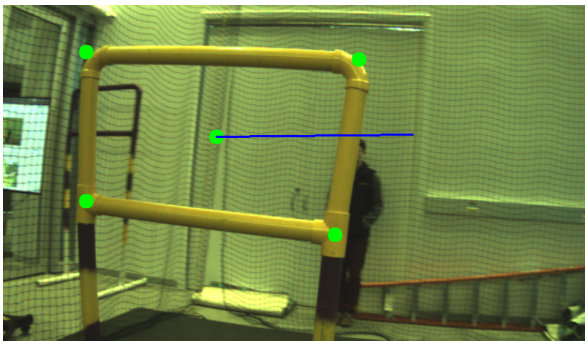

Fig. 12. Gate Pose and Orientation Determination

During the control loop the exit condition is checked and the proper controller called. The exit condition can be time based, distance based, or a check of if it has confirmed a gate. Once the exit condition is met the states is progressed to the next state number. The controller type defines the how the vehicle navigates towards its desired location. This is further explained in the next section. The $fly\_WP\_Location$ defines where the drone will try to fly if it is flying to a point in space, while the $look\_WP\_Location$ defines where the vehicle will point towards while it is flying to the point. This is set to null when the vehicle is flying towards a gate. These point are also relative so they are defined relative to the drones position when it exits the previous state. This is to counter drift in the odometry over numerous states.

When flying through a gate 3 states are used. The first states guides the vehicle to a point in space where it will have the best chance at seeing the gate. During this state the detection is active, trying to lock onto a gate location. This states is exited

once a gate location has been found. The next state works off of the gate visuals to align itself with the gate and move towards it. At a certain point the gate is no longer visible in the camera frame, at which point the states is switched to the next again. This is trigger by distance to the gate. The final state traverse the gate for a certain amount of time before the state is exited and the system is shutdown.

V. Controller

The controllers used for the system were defined in three separate functions, forward navigation, gate navigation, and point navigation. The point navigation is the same method used for project 2, and is used to fly to a position in space. This controller that uses a receding horizons approach is outlined in the paper for project 2. The forward navigation is the simplest controller as it just commands the vehicle to pitch forward at a desired angle. This controller is fully open loop and is just used to navigate through the gate. When this controller is used it is assumed the vehicle is already perfectly lined up with the gate.

The gate navigation was the designed to align the vehicle with the gate and put it on a trajectory to safely pass through. The controller executes the altitude and rotational controller independent of the lateral positional controller. If the position of the gate is found before the heading is confirmed the vehicle will rotate towards the gate and climb to the correct altitude to get a better view of gate. Once the gate heading is confirmed the lateral controller moves towards the normal line and through the gate. The basis of the lateral controller uses the same receding horizons approach as the point navigation, except the axis in which the positional error is calculated. Instead of the X and Y position error, the controller uses the distance from the gate and the angular distance from the normal line of the gate. The velocity desired towards the gate is calculated as a function of the lateral error from the normal line. The vehicle will stay 2 meters from the gate until the lateral distance from the normal line is below a threshold. The desired velocity graph of the gate navigation controller is displayed in 13.

VI. Videos

Videos of the resultant 5 successful test flights are available on YouTube:

1) Gate Test 1: https://youtu.be/BGtruLiFxVs
2) Gate Test 2: https://youtu.be/DDylRoM9s-M
3) Gate Test 3: https://youtu.be/tH4sCg$_b Ir8$
4) Gate Test 4: https://youtu.be/okbT4J4FrDQ
5) Gate Test 5: https://youtu.be/eWJlJ4Y-VXg

VII. Important Lessons Learned

One of the most significant lessons learned through our testing and development is the impact that lighting can have on the overall performance of the color segmentation. With all of our segmentation methods there was a noticeable effect on masking performance when the model was used in scenarios where the lighting conditions were different from the lighting conditions in the training data. We attempted to minimize the net effect of this by capturing data at different times of day and using the various data to train the models. However, even minor environmental differences (e.g. cloud coverage) could result in detrimental impacts to performance. One of the most effective tools to counteract these effects in real time proved to be the ability to change the camera exposure, which allowed us to offset the current environmental lighting effects and better match the model training data.

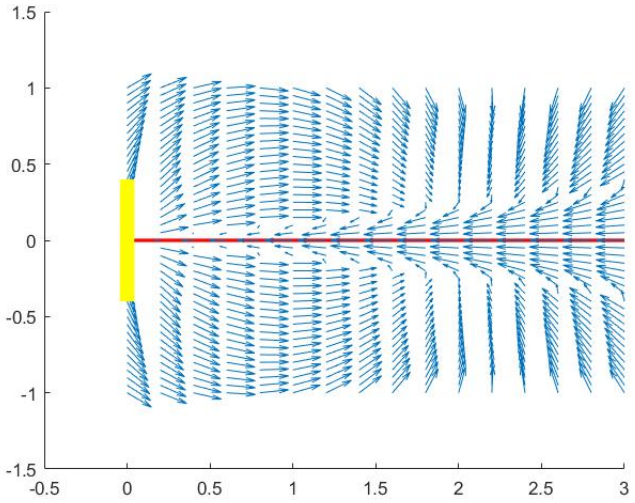

Fig. 13. Gate navigation velocity

Another factor that we discovered had significant impacts on the quality of our masked images was the resolution of the video. We had initially collected test data for training the color segmentation models at a resolution of 800x460; this included data collected at various lighting conditions and at different times of day, and multiple models trained using different color spaces. These models worked very well on the recorded data; however, the real time performance was drastically reduced. It required significant exploration of potential problems before we realized that the real time data was being captured at a higher resolution (1280x720) and that the performance of models trained at a lower resolution was impacted by this discrepancy. Updating the real time video to 800x460 resolution increased the quality of the masked images and resulted in overall improved performance.

- HAVE A SPARE UP BOARD

Acknowledgment

The authors would like to thank the professors for this course, Nitin J. Sanket and Chahat Deep Singh, as well as Dr. Inderjit Chopra.

References

[1] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.